

# USAGE OF MUTATION TESTING AS A MEASURE OF TEST SUITE ROBUSTNESS

Ronald Finkbine, Ph.D., Indiana University Southeast, [rfinkbin@ius.edu](mailto:rfinkbin@ius.edu)

## ABSTRACT

A computer program that passes its entire test suite is ready for mutation testing, which is performed by modifying a copy of the program into a mutant and executing the mutant against the same test suite. With the execution of a mutant program there can be three logical results: (1) one of the test cases fails, showing that the test suite is sufficient to detect the mutant program, (2) all test cases pass and this indicates a weakness in the test suite which needs to be repaired by the addition of a new test case, or (3) all test cases pass and a meaningless or equivalent mutation program was generated. This paper introduces mutation testing and classifies mutational operations.

## INTRODUCTION

In the area of software development, the first level of program testing is the unit test, performed by the programmer by executing their program against the collection of test cases (the test suite). This initial testing shows that a program passes its test suite but cannot assure the quality of the test suite. The mutation testing of a program attempts to measure and improve the quality of the test suite.

As a real world example, a generous faculty member can have an extremely simple examination question in an effort to give a student full credit for the question. Asking a graduate student in applied mathematics ‘What is 2 plus 2?’ is a question of a proper form, it is an answerable question, but it is an improper question if the intent is to test a graduate student of applied mathematics. In order to improve software development effectiveness, test suites need to appropriately test a subject program.

The programs that have successfully passed their initial testing at the unit level are ready for testing on the secondary level, which is mutation testing. This type of testing uses a program as an input, generates mutations of this program, and executes them to determine if the test suite detects the mutants as non-acceptable by failing a test case.

Each mutant program is executed against the test suite and there can be three logical results of this:

- One of the test cases fails, showing that the test suite is sufficient to detect the mutant program
- All test cases pass and this indicates a weakness in the test suite which needs to be repaired by the addition of a new test case
- All test cases pass and a meaningless, equivalent mutation program has been generated.

Please note the last two choices detailed above are related and it would be necessary for a programmer to interpret the results of a mutation program’s execution to determine if a meaningless equivalent mutant program was generated.

This paper introduces mutation testing, describes a mutation testing system for source programs in the Java language, and classifies mutational operations. There are three goals for this system:

- Generate mutants to detect weak test suites
- Provide a mechanism for the system designer (or analyst) to have increased control over the software developer
- Provide project platform to allow for program analysis and understanding

## RELATED WORK

Computer programs are instruction streams to the computer and the programmer but they are data to the compiler. Mutation testing is a fertile and active area in computer science research. Initially, it was used to generate good mutant programs that were faster than the original and used as heuristics in optimizing compilers [1]. From compiler optimization applications, mutation analysis has advanced to investigate test case generation [2]. Other researchers have used mutation analysis to review specifications in a design by contract implementation [3], testing network security [4], and testing distributed applications [5].

## PROJECT DESCRIPTION

For investigation and test support tool development, it is necessary to represent (a form of compiling) the subject programs, manipulate and analyze them.

The major tools in this project are:

- database storage of all programs
- database storage for all mutant modifications
- database storage for all program executions
- database storage for all test suites
- Java program for loading a java program into the database
- Java program for emitting a Java program from the database
- Java program to manage the compilation and execution of all programs
- Java compiler for compilation of generated Java programs

As background information, the equipment being used is a generic Intel PC running Windows XP. To facilitate the high priority of project transportability the database Mysql is being used as a data store and programs in the Java language are used to manipulate the data store.

Figure 1 shows the flow of the source program in its representations from source program (identified by the disk graphic) to the final product which is a table in the database (identified by the rectangular graphic) of the execution histories of the original program and its mutants against all test cases in the test suite.

## Database Design

The open source Mysql database is an enterprise-quality, open-source database (and free) available for various platforms, inexpensive to begin project development and expandable to a client-server architecture. The database tables in Figure 1 will contain representation of the program being examined and executed.

Java programs will manipulate the database, produce and execute the mutations. Each Java program is to be very simple in nature, near the deterministic finite automata (DFA) in the order of

program complexity, similar to the State Diagram of the UML [6]. This property allows this author to write programs that are *near-provable*. The definition of the term *near-provable* depends upon acceptance of diagrams as proofs, a current research area in the area of applied mathematics [7].

## Code Components

There are a number of programs in this project and each is represented in Figure 1 as an ellipse.

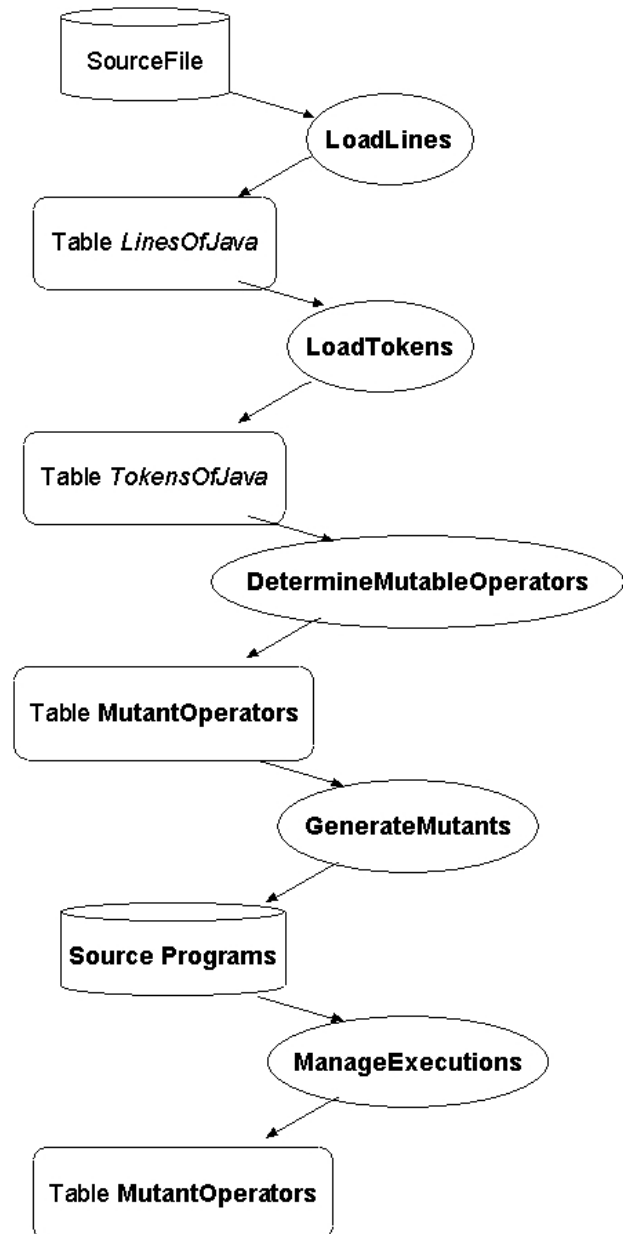


Figure 1: System Architecture

## LoadFile

This program loads a Java source file into the database table *LinesOfJava*, Each line of text has spaces trimmed from either end and each line is numbered sequentially. As an example of a DFA implementation, Figure 2 depicts the DFA for the *LoadFile* program.

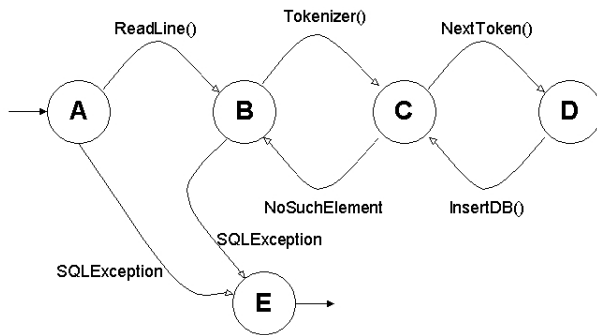


Figure 2: DFA LoadFile

This program has five states, *A* being the start state and *E* is the final state. The function *ReadLine()* executes and change the state to *B*. The DFA concept works well in linguistics and theoretical computer science as simple data acceptors. The fields of industrial electronics and electrical engineering use this concept in Mealy and Moore machines. Large software programs can be broken into streams of simpler programs implemented as DFA machines and using a database for variable storage.

## LoadTokens

The program *LoadTokens* inputs each record from table *LinesOfJava* (which represents a line of java source code) and uses the Java tokenizer package to convert each line into a sequence of tokens. Each token is placed in the table *TokensOfJava* in the database.

## DetermineMutableOperators

The program *DeterminMutableOperators* takes as input each token from table *TokensOfJava* creates a table *MutantOperators*. This program identifies the operators of the source/target program that are mutable in each of three classes, mathematic, relational and logical. The concept of

mutable operator is to replace one operator with another that will not introduce a compile-time error.

## GenerateMutants

The program *GenerateMutants* takes input from the *TokensOfJava* table and generates output files to be compiled and executed. First, the original source program will be produced with no mutation. The mutation of a source program can be viewed as a sequential traversal of all the operators in the source program (listed in the *MutableOperator* table) and is detailed in the next section.

The possible number of mutants generate d can be large and test personnel might wish to restrict this number. The tester might restrict the number of mutants produced to numbers such as: one random mutant, ten random mutants, ten mutants in sequence (of operators in the program and of the variety of operators) and generate all mutations.

The output of the *GenerateMutants* program will be details about the execution of each program, whether the original program or a produced-mutant program. Each program execution will generate a record that is placed into the table *ExecutionDetails*.

## Mutant Classification

An extensive classification of mutable operators has been performed by other researchers [8] but this project currently restricts mutable operators to the mathematical, relational and logical operators.

The mathematical operators will be in sequence: plus, minus, multiply, real division, integer division, modulus (+ - \* / %).

The six relational operators will be in sequence of equals, not equals, less than, greater than, less than equal, and greater than equal (=, !=, <, >, <=, >=).

The three logical operators will be in sequence of and, or and not (&, |, !).

## Conclusions and Future Work

This project offers a number of paths of development such as DFA-level programming, mutation testing for improving test suites, program manipulation and program understanding.

First, large software systems implemented by streams of DFA programs connected by a database

as a way of simplifying these systems provide a method of moving formal methods of program analysis and development into the mainstream of software development.

Secondly, further development of mutation testing as a method of measuring and strengthening test suites, benefits both academic and industrial software development.

Thirdly, we are pursuing development of a method program representation as a platform for program manipulation and understanding. This effort continues research for the author in the area of legacy software and algorithm recognition.

## Author Information

Dr. Finkbine is an Assistant Professor of Computer Science at Indiana University Southeast and has a Ph.D. in Computer Science from the New Mexico Institute of Mining and Technology.

## References

1. Mutation 2000: Uniting the Orthogonal, J. Offutt and R. Untch, in the book *Mutation Testing in the New Century*, edited by W. E. Wong, Kluwer Academic Publishers, ISBN 0-7923-7323-5, 2001.
2. The Dynamic Domain Reduction Approach to Test Data Generation, J. Offutt, Z. Jin and J. Pan, *Software Practice and Experience*, 29, 2 (January, 1999).
3. Trustable Components: Yet Another Mutation-Based Approach, B. Baudry, V. L. Hanh, J. M. Jezequel and Y. Le Traon, in the book *Mutation Testing in the New Century*, edited by W. E. Wong, Kluwer Academic Publishers, ISBN 0-7923-7323-5, 2001.
4. Mutation Network Models to Generate Network Security Test Cases, R. W. Ritchey, in the book *Mutation Testing in the New Century*, edited by W. E. Wong, Kluwer Academic Publishers, ISBN 0-7923-7323-5, 2001.
5. TDS: A Tool for Testing Distributed Component-Based Applications, S. Ghosh and P. Govindarajan, in the book *Mutation Testing in the New Century*, edited by W. E. Wong, Kluwer Academic Publishers, ISBN 0-7923-7323-5, 2001.
6. Sams Teach Yourself UML in 23 Hours, J. Schmuller, Sams Publishing, 1999.
7. Diagrammatic Reasoning Website, <http://zeus.cs.hartford.edu/~anderson/>
8. Mutation of Model Checker Specifications for Test Generation and Evaluation, P. E. Black, V. Okun, and Y. Yesha, in the book *Mutation Testing in the New Century*, edited by W. E. Wong, Kluwer Academic Publishers, ISBN 0-7923-7323-5, 2001.